AUDIOVERDRIVE: EXPLORING BIDIRECTIONAL COMMUNICATION BETWEEN MUSIC AND GAMEPLAY

Nils Iver Holtar, Mark J. Nelson, Julian Togelius

Center for Computer Games Research
IT University of Copenhagen
nils@nilsih.com, {mjas, juto}@itu.dk

ABSTRACT

We describe a system for bidirectionally coupling the music and gameplay of digital games, so gameplay procedurally varies the music, and music procedurally varies the game. Our goal is to inject game music more closely into the core of gameplay, rather than having the music serve as an aesthetic layer on top; traditionally, music responds to game state, but not vice versa. Such a coupling has compositional and design implications: composing game music becomes a kind of *composition of gameplay*, and furthermore, game-design decisions feed back into the music-composition process. We discuss an arcade-style 2d side-scrolling game, *Audioverdrive*, demonstrating this integrated music/game composition approach.

1. INTRODUCTION

Music is a key part of the culture and aesthetics of digital games, so much so that it often spills out of games proper, and into popular music culture. Games' soundtracks form a large part of their overall feel, and the cultural attachment they engender is such that fans flock to see symphonies perform the soundtracks [6]. Meanwhile, the particularized aesthetic sound signatures of the sound chips in systems like the Atari VCS have inspired modern-day chiptune and bitpop musicians to repurpose the hardware for music-making outside the gameplay context [4].

Despite this rich cultural spillover *outside* of digital games into a ferment of influences and cross-influences, when we examine what digital game music does back at home, in its original habitat of games, it usually plays a strangely cautious role, with only selective involvement. In games, we can locate the center of action, where the ferment of influences and cross-influences happens in *this* medium, in the closely-coupled mess made up of gameplay, interaction, and system dynamics, full of feedback, emergence, and interlocking effects. But game music typically only tiptoes around the edges of that nexus, adding an aesthetic layer to it but not getting caught in any dangerous feedback.

1.1. Game-to-music communication

The traditional way game music is coupled with gameplay is by receiving information from the core game system. The core is complex and closely coupled, but its interface with the music system is cleaner and one-way. The core gameplay system sends signals to the music system as discrete state-transition cues: The music transitions to "boss music" when the player enters a boss battle, or to "hurry up" music when a timer runs low (the states may of course be more complex, and there are often substates as well).[1]

More recently, dynamic game music aims to make this interaction more complex—but still one-way. Instead of pre-composed scores that are cued by a finite set of state transitions, gameplay events and changing game state feed into parameters of a generative-music system [2]. Intriguingly, from the perspective of our current interests, in music games such as *Guitar Hero* the influence is the other way around. There, gameplay takes input from the music: the music is in effect the "level design", specifying what the player will have to do to pass a level.

1.2. Bidirectional game–music communication

Why not throw music right into that vortex of multidirectional close coupling and feedback that makes up the heart of gameplay? That's our long-term goal: game music drilled into in the core of a game's dynamics.

In this paper, we ask something closely related yet architecturally simpler. We do maintain the nicely sealed computational boundary between "the game system" and "the music system", in part so we can reuse existing technology on each side. However, we aim to break the compositional boundary: the two systems communicate in a pervasively bidirectional manner, with neither layer treated as subsidiary. This produces a closely coupled system with complex interaction patterns and feedback between gameplay and music. Our particular interest is in treating this closely coupled system as a unified compositional situation. In a quite direct sense the composer of game music becomes a *composer of gameplay*—and in the other direction, gameplay design becomes a kind of music design.

Our contributions to enable this integrated composition process are a framework for bidirectional gameplay–music communication, and *Audioverdrive*, an arcade-style game designed and composed using the framework. The framework connects games programmed in Cocos2D, a

[1] This style of gameplay-driven, state-based music transition was pioneered by the LucasArts iMuse system in the early 1990s [12], and remains the dominant mode of game-to-music coupling [1, 3].

game engine for the iPhone and iPad platforms, with generative spaces of music composed in Ableton Live, a popular piece of digital audio workstation software aimed at live performance, via a composer/designer-configurable set of mappings. *Audioverdrive*'s aesthetics are loosely based on the first author's experiences—previously separate ones—as composer/keyboardist for the game-music-influenced synth band Ultra Sheriff, and designer of procedurally varying arcade games. One of the goals in that regard is to produce an actually playable version of the coupled gameplay–music experiences one often finds imagined in music videos in this genre.

2. BACKGROUND

Our goal of architecting the gameplay–music coupling as a bidirectional feedback loop will no doubt sound familiar to audio installation artists. In contrast to games' limited experimentation with such feedback loops, feedback between audiences and generative music systems (often conceptualized as a form of cybernetic coupling) is a common strategy deployed and explored by interactive sound installations. Therefore, in a sense our work can be seen as part of a recent trend in experimental digital games, which canvasses electronic art for ideas and design elements to selectively borrow [8].

Despite intriguing connections to audio installations and interactive electronic art more broadly, we see ourselves as situated primarily in game design as a starting point. The distinction is admittedly not a clean one [10], but we find it productive here to start with existing game-design and game-music composition practices, and experiment with adding bidirectional coupling between the two. It's possible the result may converge nearer to electronic art, especially in particular designs using feedback loops aesthetically modeled on those common in cybernetic art. But so far, in our own use of these experimental tools (see Section 4), the result still feels much like game design and game-music composition, albeit in a weirdly coupled way.

2.1. The composition-instrument

The existing style of game design closest to our goal of bidirectional coupling is probably the one theorized by Herber [7] as a *composition-instrument*. In a composition-instrument, the player can "play" or "compose" music in real-time while playing the game. In contrast to music-matching games such as *Guitar Hero*, the player generates (part of the) music through gameplay rather than matching gameplay to pre-defined music.

A particularly intriguing example from 1987 stands out, *Otocky*.[2] A sidescrolling arcade shooter for the Nintendo Famicom Disk System, it places players in control of a flying avatar that fires a short-range ball projectile in order to deal with enemies. Each of the eight possible shooting directions has a musical note attached

[2]Toshio Iwai, SEDIC/ASCII, 1987. We discuss only *Otocky* here, as a pioneer of the style; additional games are discussed elsewhere [7, 11].

Figure 1. *Otocky*, in which notes and instruments are tied to weapons.

to it, making the soundtrack directly player-influenced. *Otocky*'s playable notes always map to a note belonging to the mode and harmony in the current accompanying layer, so that player-created melodies will never contain notes that sound "wrong" or "off" from the perspective of traditional harmonic structures. Furthermore, shooting is quantized to the beat so that all notes played will fit the rhythmic structure.

The mapping here is still mostly one-directional: music is dynamically generated from player actions, but does not then feed back into the gameplay. However, musical considerations implicitly feed back into gameplay design through the constraints that were added to make "playing" produce the desired effect. This is seen most clearly in the shooting quantization. Although implemented straightforwardly in the gameplay domain as a quantization of shots, which in turn results in the notes produced by the shots being quantized, clearly the purpose of the constraint is the audio-domain quantization. It is therefore best thought of conceptually as a constraint in the audio domain, which travels "backwards" through the shot-to-note mapping to produce a constraint in the gameplay domain. The constraint here is fairly simple to hand-code in either domain, but with more complex musical constraints it is easy to see how less obvious interplay may arise.

2.2. Bidirectional procedural content generation

In addition to the experiments with dynamic and generative game audio already discussed, there has also, since the early 1980s, been work on procedurally generating game levels and other content. For example, classic games like *Civilization*, *Rogue* and *Elite* feature content that is automatically generated rather than created by a human designers. In some cases, levels are randomly generated, while in other cases a human player or designer is given some kind of control over the generated content [9]. In other cases, content is generated based on external data. In one example, Monopoly boards are generated based on demographic data for a given geographical area, using cri-

teria for inclusion specified by the player [5]. An example of procedural game level generation based on music is *Audiosurf*, where the player can supply MP3 files which are automatically analyzed by the game's software and turned into tracks for a form of racing game.

We focus on a mapping framework for the composer to manually link gameplay elements to musical elements and vice versa, taking a composition-oriented approach to the feedback loop. It is also possible to conceive of a more artificial-intelligence-oriented version of the approach: two procedural-generation systems, one game-to-music and the other music-to-game, hooked into each other in a loop, each one's output serving as the other's input. This would add a more substantial computational layer to the mappings. Where ours are quite direct, if the mappings were full-blown procedural generators, an additional algorithmic element arises, where it is not only the mappings but the algorithmic processes by which they're computed that become aesthetically and experientially relevant. On the other hand, complex mappings risk breaking the aesthetic coupling by producing such a chaotic algorithmic coupling that the player no longer sees or is able to interpret the linkage.

2.3. Ableton Live

The audio side of our mapping framework is provided by Ableton Live[3] a popular digital audio workstation (DAW) package, *i.e.* software that allows arranging and processing of audio and MIDI, along with support for third-party plugins. Live is commonly used by producers, DJs, and engineers, and is particularly focused on live audio manipulation. Most notably, it features a workflow mode called the session view, where the typical horizontal timeline view is replaced by a matrix of cells, named *clips*. Clips are grouped into *scenes* and spread across individual *tracks*. A *scene* is a row, while a *track* is a column of clips. Tracks function like in traditional DAW software, where it is an output (stereo or mono) that can be assigned a chain of DSP units. Clips host the actual content played back to the track output, and only one clip can play per track at a time. Clips can either be launched individually or in groups, the most accessible way being through launching entire scenes (Figure 2 shows an example session view).

Ableton Live's setup suits us well when working with dynamic game music, since a state change in the game can be mapped to trigger a specific scene or clip: the "live" part of the performance is here being played by our mapping application rather than directly by a human performer. Live also offers various quantization options, meaning that one can ensure that every action will be quantized to the next specified time unit. Out-of-the box, there are many MIDI routing and mapping possibilities, enabling external MIDI controllers to be configured to add a substantial amount of control. The possibility space is further expanded when the official Max For Live extension is included, as this enables the creation of custom instruments and effects. Although we don't explore it in our current experiments, the support for live audio manipulation/composition opens up the further possibility of a second person "livecoding" the game as it's being played, by manipulating the musical composition and audio parameters which are in turn mapped into gameplay.

[3]https://www.ableton.com/

Figure 2. An example of a clip matrix in Ableton Live. The small play buttons are used to launch either clips or scenes and the column to the far-right represents scenes.

3. THE MAPPING FRAMEWORK

Our mapping framework is an application that sits between Ableton Live and the game, communicating with Ableton Live over the Open Sound Control (OSC) protocol and serving as the hub of gameplay–music coupling (see Figure 3).[4] Ableton Live will report and alter its state based on received OSC messages, and the mapping application uses this functionality to create a tree list mirroring the structure of the current Live project. If the Live project is set up to explicitly send MIDI output to the mapping application (which will appear as a selectable MIDI output destination), some basic MIDI messages such as note-on and modulation will also be mappable.

Any program that exposes itself as an OSC node on the same network as the application will be selectable as a "game location". Upon selecting a game location, the mapping system will query the IP address for mappable game parameters and if successful, all mappable game parameters will be registered in the application. The composer selects which parameters to use in the current configuration. The application will in turn inform the game

[4]OSC support is added to Ableton Live by the unofficial but widely used LiveOSC plugin (`http://livecontrol.q3f.org/ableton-liveapi/liveosc/`).

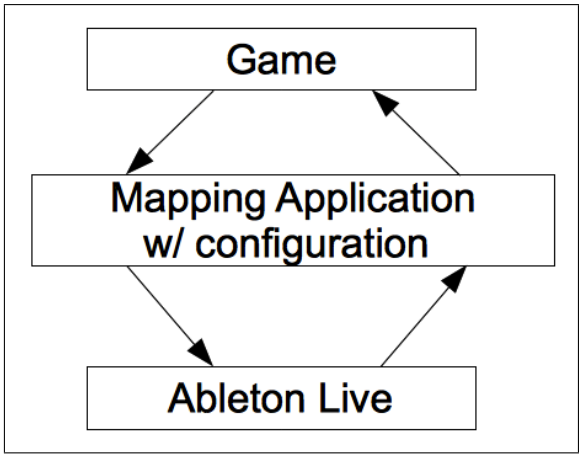

Figure 3. The mapping application and its current configuration acts as a central hub between Ableton Live and the game.

about which parameters are used, limiting the amount of OSC traffic to the bare necessities.

3.1. The anatomy of a parameter

Each parameter is a floating-point number, though arbitrary semantics can be encoded into it. It might be a scaling property only assuming values between 0.0 and 1.0, or it could be a numerical representation of a color. We also distinguish between two types of parameters from a temporal perspective: *continuous* and *discrete* parameters. A *continuous* parameter is one that updates very frequently, and should be thought of as a continuous stream of values. A *discrete* parameter is one that updates less frequently, and would usually be linked to the initiation of a process of some kind, or an event.

In Ableton Live, this distinction is very clear as continuous parameters are the equivalent of knobs or sliders while discrete parameters are usually the triggering of clips and scenes. In a game, the distinction can quickly become a little more fuzzy. A parameter that reports the amount of enemies on screen could be seen as a continuous value, but the moment in time where an increase or decrease of this number occurs could in some cases also be viewed as a discrete parameter change, or event. To simplify, we let the frequency of updates loosely dictate the parameter type. In the actual source code, there is no significant functional difference between these two; the distinction is only made in the mapping interface as a compositional aid for classifying parameters.

3.2. The anatomy of a mapping

A valid mapping consists of one or more *input* parameters and one *output* parameter. The direction of the mapping can either be from Ableton Live to the Game or vice versa. Whenever one of the input parameters changes, a function will execute, and its result will be transmitted to the output parameter. The function is a user-written javascript

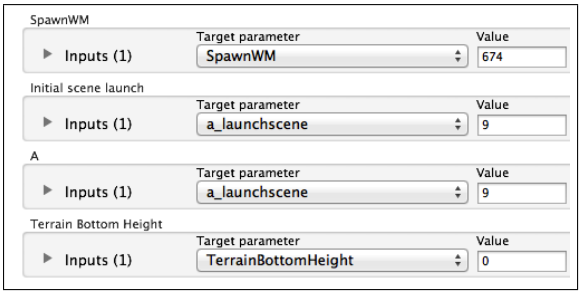

Figure 4. An example of a mapping setup.

function where the current input values are passed as arguments. The javascript instance is shared across all mappings, allowing mappings to influence each other through global variables created via the "Global Variables" button. This basic set of building blocks opens up many possibilities. The result of one mapping could for example scale the results of another. An output parameter, however, will only transmit its value when triggered by an input parameter, so it is not possible to directly trigger the calculation of one mapping from another.

4. AUDIOVERDRIVE

Audioverdrive is an iPad game, using the Cocos2D game engine. Ableton Live and the mapping software run on an OS X computer, communicating with the iPad app using OSC over a wireless network. The game is inspired by the genre of space shooters that includes *Gradius*[5] and *PixelJunk SideScroller*.[6] The player controls a side-scrolling ship, attempting to avoid collision with enemies, enemy bullets, and terrain. Also present in the world are weapon-modifying orbs that the player can either absorb or activate through proximity. The orbs come in four varieties: power, spread, homing, and bounce. When activated, an orb's property will be added to the ship's weapon, combining itself with any properties already added. Should the player choose to absorb an orb by tapping it, that orb's property will be added to the ship permanently—or until the player absorbs a different orb, the ship is hit by a bullet, or the ship is destroyed.

4.1. The game parameters

When run without OSC communication, little happens. The player can control the ship and pick up weapon orbs, but since no enemies are spawned (the default enemy spawn rate is zero), there is no incentive for the player to interact. An analogy could be made in viewing this state of the game as a synthesizer waiting to be played. Not surprisingly, the game takes a turn for the more interesting once the game parameters are put into use.

Table 1 lists all input game parameters that *Audioverdrive* accepts over OSC from the mapping application.

[5] Konami, 1985

[6] Q-Games, 2011, `http://pixeljunk.jp/library/SideScroller/`

Player speed	Continuous
Enemy Speed	Continuous
Terrain Speed	Continuous
Player Shoot	Discrete
Enemy Shoot	Discrete
Enemy Mode	Discrete
Enemy Spawn Rate	Continuous
Spawn Enemy	Discrete
Weapon Orb Type	Discrete
Weapon Orb Spawn Rate	Continuous
Spawn Weapon Orb	Discrete
Terrain Top Height	Continuous
Terrain Bottom Height	Continuous
Color None	Continuous
Color Spread	Continuous
Color Homing	Continuous
Color Power	Continuous
Color Bounce	Continuous
Color BG	Continuous
Color Terrain	Continuous
Game Victory	Discrete
Game Defeat	Discrete

Table 1. Available input parameters for the game.

Some instantiate objects in the world, while others affect the behavior of the world and objects already in it. This was done primarily with the intent of enabling the creation of gameplay curves in tandem with musical curves. Changes in these parameters are also instantly perceivable in the game. So while it is still up to the composer to decide the degree of directness between the audio mappings and the parameters that govern the game world, the parameters exposed were chosen so as to make for a high chance that relations will be recognizable by the player, if anything clearly audible is mapped to these parameters.

We have also chosen to expose the victory and defeat actions, the consequence of this being that the composer essentially controls what constitutes victory and defeat in the game. Making the end conditions trigger in interesting ways, however, is only achievable through clever utilization of both these input parameters and the output parameters depicted in Table 2. These values, received from the game and transmitted by the mapping framework to Ableton Live, should be made to influence the music in ways that change the way the music again influences the game. This is where the composer can begin to really experiment, as it is possible to effectively create new rules of the game this way (illustrated in the next section).

While the novelty of our framework lies in its possibility for creating system-altering mappings, some color parameters are also manipulatable. This was done in an attempt to provide composers with more freedom in regards to which musical style to employ. It was our goal to avoid the potential restrictions that a too-predefined aesthetic might introduce. Also, since there is no screen overlay showing points, lives remaining, ammunition, or other features often included in a game's heads-up display, the color parameters are also the composer's only tool for offering other kinds of visual feedback that go beyond instantiation of game objects.

Player X Position	Continuous
Player Y Position	Continuous
Player Deaths	Discrete
Total onscreen enemies	Discrete
Total destroyed enemies	Discrete
Enemy1Shot	Discrete
Number of spread	Discrete
Number of homing	Discrete
Number of power	Discrete
Number of burst	Discrete
Current modifier	Discrete

Table 2. Available output parameters for the game.

While the parameters exposed here are a relatively small set of those that can be imagined even for just this style of game, our experience is that even these make for a very configurable game, enabling a wide range of possible designs. We especially embrace the possibility that this can yield very unusual results seen from a traditional game design perspective, since composers may choose to favor musical qualities, rather than designing from a gameplay-first perspective at all times. A musically unique bassline might be chosen in preference to a finely tuned difficulty curve, for instance.

4.2. The game design

The game design is broken into three parts: the music-to-game mappings, the game-to-music mappings, and finally, an overall phase structure in which both the gameplay and music progresses from an early phase through several intermediate phases, to the culmination of the level. This phase structure can be seen as both the "level design" and the "composition", the composer's choices guiding how the coupled experience unfolds at the macro scale.

4.2.1. How music controls the game

The musical aesthetic is an electronic track inspired by acts like *Daft Punk* and *Lazerhawk* and some elements of soundtracks to 1980s space shooter games. The instrument lineup is fairly basic, with relatively direct mappings to game state. Enemy shots are triggered when a kick or a snare drum plays, and all enemies currently on screen will attempt to crash into the player when a clap sample plays. A distorted sitar instrument is mapped to spawn enemies, and this usually happens at the beginning of four-bar periods, backed up with a crash sound. A short plucking instrument triggers player shots, usually in rapid succession, as this instrument mainly plays quick arpeggios. The terrain height is controlled by the current note value of

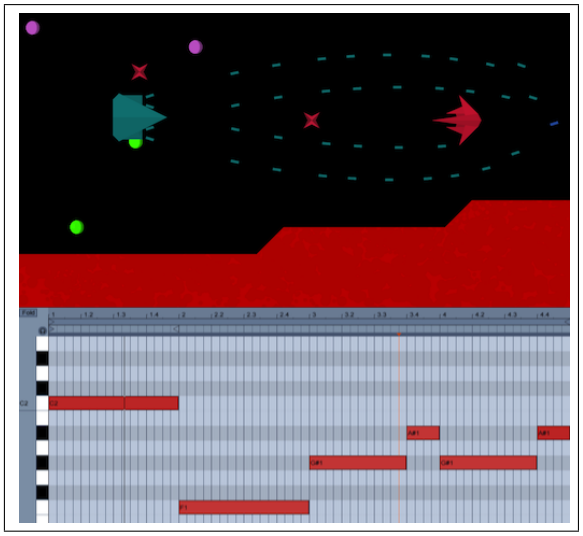

Figure 5. The lower image shows the voice of deep pad bass clip currently looping when the screenshot was taken. The two last notes lead up to the first note in the loop (Ab-Bb-C) which is what just happened in the screenshot, as can be seen in how the bottom terrain rises in steps.

a deep pad sound (which will be referred to as the *terrain instrument* for convenience), and this instrument is played with two separate voices, where the lower voice controls the bottom height and the upper voice controls the top height of the terrain. Figure 5 illustrates how the terrain and instrument can correlate. The mapping is made so that smaller intervals between the voices result in narrower passages in the game. Weapon orbs are spawned every time a sonar-like instrument (made up of white noise and several sine waves) plays, where the *y* position of the spawned orb is decided by the note pitch.

4.2.2. *How the game controls the music*

The game controls the music mainly by triggering clips and scenes in Ableton Live when events occur that resulted from player actions (directly or indirectly). Since the clips in turn instantiate and modify game elements when played, this produces the bidirectional communication. For example, when enemies are destroyed, an explosion clip is triggered along with a three-note chord played by the sonar-like instrument. Due to its mapping, this chord will also instantiate weapon orbs in the game state. That demonstrates how bidirectional mappings can work together to create new rules in game, since the instantiation of weapon orbs becomes a direct cause of destroying enemies. Different clips will be launched in Ableton Live depending on the current set of orbs connected to the player. The orbs have been split into two groups, where the bounce orb is matched against the terrain instrument and the other orbs are matched against the arpeggio instrument. Exactly which clip is launched differs depending on the number and type of orbs linked to the player. Picking up or connecting with different orbs therefore has different effects in the game; bouncing orbs will indirectly change the terrain, while other orbs will influence the player's shot patterns.

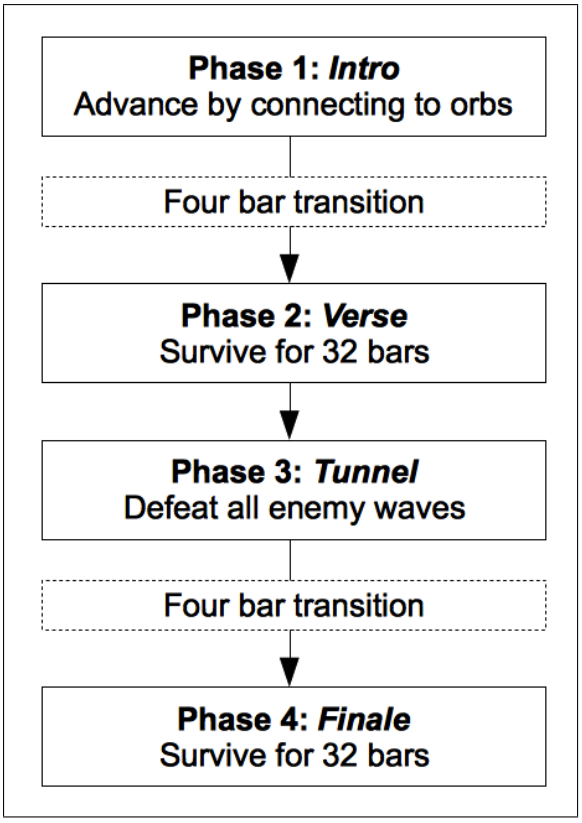

Figure 6. Gameplay phases in *Audioverdrive*.

4.2.3. *Composition of the gameplay–music progression*

The game is structured in four sequential phases (see Figure 6). Phases are programmed in the mapping framework (in Javascript), and influence which clips and scenes in Ableton Live, and therefore which gameplay sequences, are triggered. The track is built around one short musical motif, with phases driving the musical progression built from this motif, roughly in ABAB form. The player's goal is to survive until the end, which can only be achieved by completing the challenge in each phase.

The first phase starts with a simple two-note sonar clip creating bounce weapon orbs. Upon connecting with an orb, a terrain instrument clip is triggered. Connecting with more than one orb replaces the clip with more lively versions, resulting in more jagged terrain. The goal in this phase is to succeed in maintaining a connection of two or more orbs for a period of four bars each time. With each success, a crash sound will play while new instruments are gradually added and existing clips are replaced. This phase is meant to offer some exploration of the terrain generation, while also gradually introducing the main musical motif. The two final crash cymbals in this phase are also accompanied by the enemy-spawning sound. At this point in the track, neither the arpeggio nor the drums play, so no enemy or player shots are fired. However, a clap appears at the end of each four-bar period whenever

an enemy is present, causing the enemy to charge towards the player. When the player has succeeded in maintaining sufficiently long connections, the track automatically builds up and transitions into the next phase.

In phase two, the player needs to survive for 32 bars. At this point, the track has the full set of drum clips playing, resulting in enemies shooting regularly. Since the enemy spawn sound plays every fourth bar, the player must actively dodge enemy bullets while trying to connect with weapon orbs that activate the arpeggio clips, which in turn cause the player to fire shots back. Musically, this is where the track has really begun in full, with a steady bass pattern and drum beat being the main focus. Bounce orbs are omitted from the possible spawned orb types, so no terrain generation occurs in this phase, in order to create a clear contrast with the previous phase by making this phase more spatially open and focused on combat.

After 32 bars, the player will enter phase three. Similar to the second phase, the music here will not advance unless the player completes a series of goals. Here, the player needs to destroy each incoming wave of enemies, and new waves will not present themselves until the current one has been dealt with. Each enemy wave is spawned with an increasingly elaborate melody being played by the enemy-spawning instrument. Musically, this phase is a slightly more intense version of the motif played in the intro, with the terrain instrument being in the foreground. Bounce orbs are also spawned, offering the player the possibility to influence the terrain generation again. This offers a strategy for dealing with some enemies, as more jagged terrain increases the likelihood of them colliding with it. When the final wave of enemies is dealt with, the track transitions into the last phase.

Musically, the final phase is a full realization of the second phase, and is intended to be the peak of the game experience. As in the second phase, the player must survive for 32 bars. A main difference is that the sonar instrument plays much more elaborately, making the screen home to many more weapon orbs. This was done in an attempt to make the actual game experience match the climatic nature of the music, as the screen is likely to be filled with elaborate player and enemy bullets. As in the second phase, no bounce orbs are spawned, and therefore there is no terrain.

5. DISCUSSION

In our experience using this mapping framework to design/compose *Audioverdrive*, the music-composition and gameplay-design processes are indeed intertwined (as intended), though the unified composition situation often ends up posing rather challenging constraints. Some musical choices are restricted by gameplay considerations when in the context of particular mapping choices, and in the other direction, some gameplay ideas must be scrapped because of their musical effects. On the other hand, certain combinations have a synergy where gameplay patterns suggest development of a musical theme and vice versa. For example, the terrain instrument played a much more minor part in our initial design, but was expanded to use more elaborate melodies because of how well the coupling with terrain generation worked.

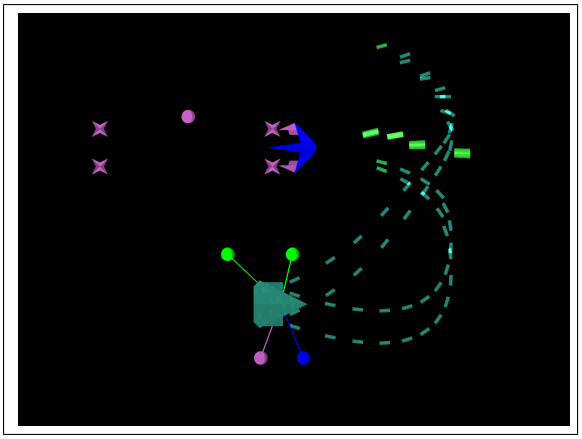

Figure 7. Screenshot from phase four, where the player must simply survive for 32 bars. The sonar instrument plays a full melody here, resulting in many weapon orbs; the player is shown connected to four of them.

A main challenge with our current approach to game–music mapping is the lack of structure given by this very open-ended space of possible mappings. Our framework currently gives maximum flexibility, simply exposing input and output parameters and allowing the composer to build a coupled experience out of them by overlaying any kind of coupling structure they wish. This constitutes a fairly low-level interface, however. The higher-level structure that a composer can build on top of the framework can take many forms, which are not reified in the interface or architecture. Therefore, to achieve coherent and interpretable results, the composer/designer needs to develop a concept or method, more specific than the space of all possible couplings, with which they'll organize the game–music mapping in their particular piece.

The particular method we used for *Audioverdrive* was to begin with a musical idea, which we used as a point of departure for organizing the complete gameplay experience. This led to a macro-level composition oriented around movements or phases, with transitions between them triggered in a fairly conventional fashion by completion of gameplay objectives. Therefore these musical phases correspond one-to-one with gameplay phases. More complex bidirectional feedback is then instantiated *within* each phase, with game–music mappings depending on the particular phase. The development of the mappings within each phase took place in a more iterative fashion, experimenting until we eventually reached a fixed point where both the gameplay and the music "worked".

The approach we took with *Audioverdrive* is only one way of approaching the composition/design of these coupled experiences. But, it may be worth focusing on and developing a few such specific approaches in more depth, to understand the composition/design of these coupled ex-

periences in a more structured context. In addition to allowing us to gain further compositional/design experience with specific idioms, a narrowed focus can also lead to opportunities for better computational and interface support for mappings. For example, we are considering building interface modes around reified mapping patterns, such as including the concept of a "phase" in the mapping interface.

6. CONCLUSION

Our goal is to push game music into the core of gameplay through strong bidirectional coupling, where music procedurally varies gameplay, and gameplay procedurally varies music. With such a feedback loop, music composition can no longer be treated as a separate aesthetic layer accompanying game-design, but is instead unified into a single compositional/design situation: composing game music composes gameplay, and designing gameplay designs game music.

In our initial work to realize coupled gameplay–music composition, we built a mapping framework that allows a composer to link gameplay parameters with Ableton Live elements in both directions. We designed a space-shooter game, *Audioverdrive*, using this framework, exploring a particular compositional style oriented around a linear progression of "phases", with coupled interaction, through fairly direct mappings, within each phase.

On the experiential side, our goal is to produce an experience that is somewhere between the composition-instrument of Herber [7] on the one hand, where the player feels they are producing music; and the music-following experience of games such as *Guitar Hero*, on the other, where the player feels they are following along with pre-written music. One of our touchstones in aiming at a coupled experience beween those two is the frequently imagined—but not actually playable—interaction between gameplay and music depicted in the music videos of some electronic-music groups.

On the composition side, our main interest has been in exploring the results of constraints in each direction, ensuring as much as possible that we are not simply layering gameplay on top of music or the reverse, but tackling their coupling. Though the game is fairly simple, we think that is the case in *Audioverdrive*. From a strictly technical perspective, in fact, the game cannot operate without both sides of the equation. The Ableton Live setup and *Audioverdrive* run on physically separated devices, connected through a wireless network. When the connection is interrupted, the game and music both clearly flounder as their animating counterpart disappears, leaving a ship floating aimlessly through space, accompanied by some monotonous synth pads.

7. ACKNOWLEDGEMENTS

Thanks the makers of modular software we strung together: Ableton Live, VVOSC, LiveOSC, Cocos2D, and Box2D.

8. REFERENCES

[1] K. Collins, *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design*. MIT Press, 2008.

[2] ——, "An introduction to procedural music in video games," *Contemporary Music Review*, vol. 28, no. 1, pp. 5–15, 2009.

[3] P. J. Crathorne, "Video game genres and their music," Master's thesis, University of Stellenbosch, 2010, http://hdl.handle.net/10019.1/4355.

[4] K. Driscoll and J. Diaz, "Endless loop: A brief history of chiptunes," *Transformative Works and Cultures*, vol. 2, 2009, http://dx.doi.org/10.3983/twc.2009.0096.

[5] M. G. Friberger and J. Togelius, "Generating interesting monopoly boards from open data," in *Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games*, 2012, pp. 288–295.

[6] C. Furlong, "Computer game music: A multifunctional medium," Master's thesis, University College Dublin, 2006, http://www.cianfurlongmusic.net/Music/Computer_Game_Music_-_A_Multifunctional_Medium.pdf.

[7] N. Herber, "The composition-instrument: Musical emergence and interaction," *Hz*, vol. 9, 2007, http://www.hz-journal.org/n9/herber.html.

[8] A. Hieronymi, "Playtime in the white cube: Game art, between interactive art and video games," Master's thesis, University of California, Los Angeles, 2005, http://ahieronymi.net/pdfs/ahieronymi_playtime.pdf.

[9] R. Khaled, M. J. Nelson, and P. Barr, "Design metaphors for procedural content generation in games," in *Proceedings of the 2013 ACM SIGCHI Conference on Human Factors in Computing Systems*, 2013, pp. 1509–1518.

[10] O. Leino, "Re-conceptualising the play-element in electronic art," in *Proceedings of the 17th International Symposium on Electronic Art*, 2011, http://isea2011.sabanciuniv.edu/paper/re-conceptualising-play-element-electronic-art.

[11] M. Pichlmair and F. Kayali, "Levels of sound: On the principles of interactivity in music video games," in *Proceedings of the 2007 Digital Games Research Association Conference*, 2007, pp. 424–430.

[12] W. Stranck, "The legacy of iMuse: Interactive video game music in the 1990s," in *Music and Game: Perspectives on a Popular Alliance*, P. Moormann, Ed. Springer, 2013, pp. 81–91.